2022

ORTHOGONAL RANGE SKYLINE QUERIES

Saano Murembya
Michigan Technological University, semuremb@mtu.edu

ORTHOGONAL RANGE SKYLINE QUERIES

By

Saano Murembya

A THESIS

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2022

This thesis has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Computer Science.

Department of Computer Science

Thesis Advisor:	*Dr. Yakov Nekrich*
Committee Member:	*Dr. Zhenlin Wang*
Committee Member:	*Dr. Ali Ebnenasir*
Department Chair:	*Dr. Linda Ott*

Contents

List of Figures . . . . . **vii**

List of Tables . . . . . **ix**

Abstract . . . . . **xi**

1 Introduction **1**

1.1 Previous and New Results 5

2 Preliminaries and Definitions **9**

2.1 Models of Computation 9

2.1.1 RAM Model 9

2.1.2 I/O Model 10

2.2 Orthogonal Range Reporting 11

3 Data Structures and Techniques **15**

3.1 Augmented Cartesian Trees 15

3.1.1 Three-Sided Range Reporting With Augmented Cartesian Trees 19

3.2 Range Trees 22

3.3 Nested Interval Trees 24

3.4 Reduction to Rank Space 26

3.5 The Grid Approach 27

4 The Range Skyline Problem 35

4.1 Orthogonal Range Skyline Queries 36

4.2 I/O Optimal Categorical 3-sided Skyline Queries 38

5 New Work on Skyline Queries 45

5.1 Novel Reduction from 3-sided Skyline Point Queries to 3-sided Orthogonal Point Reporting 46

5.2 Counting Colors in a Skyline 49

5.3 Potential Research Directions 51

5.4 Open Questions 53

References 55

List of Figures

3.1 Case (a) of the grid approach. The green and red regions can be treated as 3-sided queries with respect to the middle region. To query the middle region, we can make use of our top-level data structure. 29

3.2 Case (b) of the grid approach. The query in red lies completely within a column. To query, we can recur on the data structure assigned to this column. 30

5.1 Two cases of 3-sided skyline queries 47

(a) Case 1: The skyline is entirely above $y = c$ 47

(b) Case 2: The skyline dips below $y = c$ 47

List of Tables

1.1 Runtimes/space complexities of some existing algorithms. d is the number of dimensions in the query range, k is the number of points in the query range, n is the total number of points. B is the word size being used (only relevant in the I/O model, defined later). u is the size of the universe spanned by the points, defined later. 6

1.2 Previous and new results on 3-sided Categorical Skyline Queries, External Memory Model. QT stands for Query Time.k- number of colors in the query range 7

1.3 Previous and new results on 3-sided categorical skyline reporting with counting, RAM model. k- number of colors in the query range . . . 7

Abstract

Given a set of points P, we often need to report the ones that lie within a certain query range Q. This is referred to as orthogonal range reporting. We can also go further, reporting only the dominant points within that query. In 2 dimensions, a point $p1 = (x1, y1)$ dominates a point $p2 = (x2, y2)$ iff $x1 \geq x2$ and $y1 > y2$ or $x1 > x2$ and $y1 \geq y2$. The set of all dominant points within a query range is called the skyline of that query.

There are several different variants of skyline queries. For example, we can consider each point in P to be colored. Given a query range Q, can we efficiently count the number of points of each color in the skyline? In this thesis, we will present a new $O(\frac{\log n}{\log \log n} + D \log \log n)$ method for doing so. The method is possible thanks to a new reduction from skyline queries to orthogonal range queries. We will also explore novel algorithms for answering skyline query variants in the I/O model of computation, making use of techniques such as Ganguly et al.'s [2] double-chaining method and Alstrup et al.'s [14] grid approach. By applying these existing techniques in new ways, we can not only derive our own efficient algorithms for skyline queries, but also explore potential avenues for future research.

Chapter 1

Introduction

In the age of big data, the sheer size of datasets often obscure the most relevant information. For example, in many optimization problems, optimal points of data are surrounded by many suboptimal points. The set of optimal points is often referred to as a skyline. Given a set of t-dimensional points $(x_1, x_2, ..., x_t)$, the skyline of the set consists of only those points that are dominated by no other points. A point dominates another iff it has better values for all dimensions. For example, consider a set of houses, represented by 2-dimensional tuples (price, age). House X dominates house Y iff X is cheaper AND newer. If you were buying a house, there'd be no need for you to consider house Y because it's older AND more expensive. As a result, house Y wouldn't be in the skyline of this set of houses.

A naive method to find the skyline is searching through and comparing every pair, an $O(n^2)$ operation. In their paper on the topic, Borzsonyi, Kossman, and Stocker [13] sought to do better. They proposed the addition of a Skyline operator to SQL, built upon a relatively basic divide-and-conquer algorithm. Borzsonyi et al. proposed computing the median value of some dimension x_t, partitioning the two points around this median, computing the skyline of each subset, and finally merging the two skylines into one. With this algorithm, they're able to achieve a complexity of $O(n \log^{d-2} n) + O(n \log n)$, where d is the number of dimensions in the skyline. This is a significant improvement over the naive $O(n^2)$ algorithm

Nevertheless, the algorithm proposed by Borzsonyi et al. suffers from multiple shortcomings. Firstly, this algorithm doesn't use any preprocessing to its advantage. Each iteration of the algorithm starts with all the data completely unorganized in an SQL table. Secondly, Borzsonyi et al.'s algorithm doesn't take the size of the skyline into account. For instance, if a given data set only has 1 point in the skyline, surely there's no need to execute an $O(n \log n)$ algorithm just to find that 1 point, right? What if $n = 100000$, for example?

Perhaps we could take advantage of Bernard Chazelle's filtering search algorithms [5]. Here, Chazellle tackles the abstract problem of filtering a large dataset for points that satisfy some query, then applies these abstract principles to several concrete problems. Chazelle was able to achieve asymptotic time complexities that depend

upon the size of the query. For example, one of the problems that Chazelle addresses is the interval overlap problem; given a set S of intervals, report all the ones that intersect a query interval q. Using a filtering search algorithm, Chazelle is able to achieve an $O(k + \log n)$ complexity, where n is the size of S and k is the number of intervals in S that intersect q. Using Chazelle's algorithm, we can construct efficient skyline queries that take the size of the skyline into account.

However, Chazelle's algorithm, while groundbreaking at the time, has since been superseded by better algorithms. For instance, Alstrup et. al [14] revisted Chazelle's paper 15 years after it was originally published. Within, they detailed how Chazelle's idea of optimizing queries for the size of the query could be applied to orthogonal range queries. Orthogonal range queries are defined as follows: given a set of points P in $\mathbb{R}^d$, find all points that are contained within an orthogonal range Q. An orthogonal range is a d-dimensional rectangle $[a_1, b_1] \times ... \times [a_d, b_d] \subseteq \mathbb{R}^d$. Alstrup. et al managed to surpass many of Chazelle's algorithms for reporting orthogonal range queries. For instance, while Chazelle was able to achieve $O(k + \log n)$ runtime for two-dimensional range reporting, Alstrup et. al were able to achieve $O(k + \log \log n)$ runtime.

However, Borzsonyi et al.'s algorithm was only designed to report general skylines. Perhaps we would rather report a range skyline. The general skyline problem seeks to find a skyline in a dataset, but the range skyline problem only considers points in a certain range. The range skyline is far more applicable to realistic situations.

In the range skyline problem, each point not only has a $(x_1, x_2, ..., x_t)$ coordinate of t dimensions, but also an $(x_1, x_2, ..., x_d)$ coordinate of d dimensions. The skyline is still calculated using $(x_1, x_2, ..., x_t)$, however, only points within a certain part of d-space are considered. The increased power of this can be illustrated by referring back to our house example from before. In the general skyline problem, House Y will never be considered because it's cheaper and older than House X. We might as well just discard it from the database. However, in the range skyline problem, we not only consider (price, age) of each house, but also (latitude, longitude). The range skyline problem involves getting the skyline of houses within a certain lat/long range: $[\text{latitude}_1, \text{latitude}_2] \times [\text{longitude}_1, \text{longitude}_2]$. As you can see, this is far more useful. What if House Y is in Michigan, but house $\times$ is in Texas? If you're buying houses in Michigan, you wouldn't be eager to buy house X, even though its cheaper and newer than house Y.

In the decades following Borzsonyi et al.'s paper, numerous other works have sought to address the range skyline problem. For example, Rahul and Janardan [12] take advantage of preprocessing, coagulating the original set of points into data structures that can be queried for range skylines in much faster than $O(n \log n^{d-2}) + O(n \log n)$ time.

A more recent paper by Ganguly, Gibney, Thankachan, and Shah [2] accounts for the fact that in modern machines, memory accesses are the primary bottleneck to speed.

In their paper, they sought to process a data set for range skyline querying in a way that minimizes memory access time. They did so by designing their algorithm in the I/O model of computation.

To fully explain what contributions this thesis makes to the field, we'll first have to explain the underlying concepts. These concepts include orthogonal range searching, the grid approach, the I/O model of computation, and more. Given the research that has already been put into this problem, we will have to go beyond basic data structures to understand these concepts. Advanced data structures that will be used and explained include the range tree, the augmented Cartesian tree, and the nested interval tree. With all these preliminaries in place, we'll be in a position to explain how the new reduction from skyline queries to point queries works. From there, we can detail the novel $O(\frac{\log n}{\log \log n} + k \log \log n)$ skyline color reporting algorithm that's based on this reduction. Finally, we'll make use of these existing concepts to explore potential directions of future research.

1.1 Previous and New Results

Theorem 1 Let $b' = max\{p.x | p.y > c \text{ and } p.next > b\}$. For any point p, p is on the skyline of the query range $[a, b] \times [c, \infty]$ iff $a \leq p.x \leq b_1$ and $p.next > b$, where $b_1 = min(b, b')$.

This theorem can be used to reduce the skyline problem to an orthogonal range query. Some previous work tries to tackle the skyline problem directly, but this thesis details how it can instead be reduced to the simpler orthogonal range query. Full details are given in Chapter 5.

Theorem 2 For query $Q = [a, b] \times [c, \infty]$ on a set of points P, there's a data structure that can report the number of points of each color on the skyline of Q, taking $O(\frac{\log n}{\log \log n} + k \log \log n)$ t time where k is the number of colors in the range. This data structure can store the set of colored points in $O(n \log \log n)$ words of space.

This theorem describes the new $O(\frac{\log n}{\log \log n} + k \log \log n)$ method for reporting the number of points of each color in the skyline. Full details are given in Chapter 5.

Table 1.1
Runtimes/space complexities of some existing algorithms. d is the number of dimensions in the query range, k is the number of points in the query range, n is the total number of points. B is the word size being used (only relevant in the I/O model, defined later). u is the size of the universe spanned by the points, defined later.

Data Structure	**Ref**	**Query Time**	**Space Usage**
d-dimensional Orthogonal Range Reporting	Range Trees	$O(\log^d n + k)$	$O(n \log^{d-1} n)$
2-dimensional Orthogonal Range Reporting	Alstrup et al. [14]	$O(k + \log \log n)$	$O(n \log^{\epsilon} n)$
3-sided Orthogonal Range Reporting	Alstrup et al. [14]	$O(k)$	$O(n)$

Table 1.2
Previous and new results on 3-sided Categorical Skyline Queries, External Memory Model. QT stands for Query Time.k- number of colors in the query range

Reference	**QT on $u \times u$ grid**	**Space Usage**
[2]	$O(k/B + \log\log_B u)$	$O(n \log^* n)$
This thesis	$O(k/B + \log\log_B u)$	$O(n \log^* n)$

Table 1.3
Previous and new results on 3-sided categorical skyline reporting with counting, RAM model. k- number of colors in the query range

Data Structure	**Query Time**	**Space Usage**
3-sided categorical skyline reporting w/o counting[2]	$O(1+k)$	$O(n \log^* n)$
3-sided categorical skyline reporting with counting [2]+[8]	$O((\frac{\log n}{\log\log n})^2(k+1))$	$O(n \log n)$
3-sided categorical skyline reporting with counting (This thesis)	$O(\frac{\log n}{\log\log n} + k \log\log n)$	$O(n \log\log n)$

Chapter 2

Preliminaries and Definitions

2.1 Models of Computation

2.1.1 RAM Model

The RAM model of computation is what's most commonly used when analyzing algorithmic complexity. Modern computer architecture can have significantly different speeds for different operations. The superscalar width, ALU speed, branch prediction mechanism, and much more can impact instruction speed. By analyzing algorithms with the RAM model, we are able to abstract away all this idiosyncratic complexity. The RAM model assumes that basic operations like load/store, arithmetic operations,

and conditionals can all be done in constant time, denoted $O(1)$. Analyzing the time complexity of a program is therefore a matter of counting how many of these basic operations there are.

2.1.2 I/O Model

In the past, computer speeds tended to be bottlenecked by the CPU. However, thanks to improvements in bitwise parallelism, superscalar processing, and transistor density, this is no longer the case. In modern machines, speed is bottlenecked by memory accesses. All of the above advancements have come with an exponential increase in memory size. To effectively query this memory, we need some sort of hierarchical structure. The top layers of this structure, known as caches, are significantly smaller than the bottom layers of memory. Cache replacement policies try to ensure that we have as much relevant memory in the top layers at all times.

In this theoretical algorithmic analysis, we can't tailor algorithms to individual, real-life machines. That would require formally studying the architecture of those machines. However, we can come up with a simplified abstraction of modern cache hierarchies, often referred to as the External Memory Model. In the External Memory Model, we model our machine as having two layers of memory. The main memory (cache) has a limited size but can be queried instantly. The external memory (disk)

has an unlimited size, but it costs us a significant amount of time to query it. This time is directly proportional to B, where B is the size of the memory block being queried in words.

In the RAM model, an optimal query algorithm runs in time $O(k)$, where k is the number of points in the query range. But in the I/O model, optimality has a slightly different definition. The complexity of the program is measured as the number of I/Os with a word size B, so an I/O optimal query algorithm runs in time $O(k/B)$. In the following sections, we will refer to papers about I/O optimal skyline queries. Papers which don't consider I/O optimality in their queries have space and runtime as functions of k and n, where k is the number of points in the query and n is the total number of points. However, the papers that touch on I/O optimality will quantify their space and runtime in terms of B as well, where B is the block size defined above.

2.2 Orthogonal Range Reporting

Range reporting is an extremely integral part of many modern range skyline algorithms. This is a natural consequence of the generality and abstractness of range reporting. Given a set of points P in $\mathbb{R}^d$, we seek to find all the points that intersect with a query Q, also in $\mathbb{R}^d$. Extending upon the example given in the abstract, let's imagine that $P \in \mathbb{R}^2$ is a set of houses, each with a given price and age. If we want to

find all houses 5-10 years old with a list price $500,000 and $600,000, we could make use of a range reporting algorithm for $\mathbb{R}^2$.

Of course, range reporting algorithms can extend far beyond $\mathbb{R}^2$. By reducing the range skyline problem to a range reporting problem, we can use a wealth of established data structures and algorithms for range reporting in order to solve the range skyline problem for any arbitrary dimension.

Orthogonal range reporting is a special case of range reporting. In orthogonal range reporting, we are still dealing with a query $Q \in \mathbb{R}^d$ and a set of points $P \in \mathbb{R}^d$. But this time the query Q is an axes-parallel box. A 2-dimensional axes-parallel box is a rectangle, a 3-dimensional axes-parallel box is a cube, and this concept extends to higher dimensions. More formally, $Q = [a_1, b_1] \times ... \times [a_d, b_d] \in \mathbb{R}^d$.

Orthogonal range reports are more easily applicable to real-world situations in database querying. Databases are often structured as collections of samples, where each sample contains d features. In querying this database, we often want the points that fall within certain ranges for all the features. That's an orthogonal range query. The above house example I gave is an orthogonal range query. Focusing on orthogonal range queries has other advantages as well. In the general range query, Q could be any hyper-dimensional polygon. But in orthogonal range queries, Q is a hypercube, which is much easier for us to conceptualize. we are more easily able to apply data structures like the ones laid out in the previous section. Algorithmic motifs such as

recursion become much cleaner and much less riddled with edge cases.

A naive approach for Orthogonal Range Reporting requries no preprocessing but $O(nd)$ time; simply iterate through all the points and check if they lie in all d ranges given by Q. But as detailed later in this paper, preprocessing the points into a range tree allows us to answer orthogonal range queries in $O(\log^d n + k)$ time, where k is the number of points to be reported. Alstrup et al. [14] do even better than that. For the two dimensional case, they outlined how to answer queries of the form $[a, b] \times [c, d]$ in $O(k + \log\log n)$. To achieve this almost constant time complexity, they make use of several techniques, including reduction to rank space, grid searching, asymmetric communication-based data structures to resolve the existential range query, and more.

Alstrup et al. also tackled d dimensional queries in an abstract, induction-type manner. Consider an arbitrary data structure for reporting d dimensional queries that uses space $O(s(n))$ and time $O(t(n) + k)$. Alstrup et al. argue that one can first construct a tree with n leaves, one for each point. Each parent node points to an instance of the above arbitrary data structure. Given a $d + 1$ dimensional point, orthogonal range queries can be answered by methodically querying $O(\log n)$ of the d dimensional data structure clones. Therefore, given a data structure for two dimensional range reporting, that uses $O(s(n))$ space and $O(t(n) + k)$ time, it can be extended to a data structure for d dimensional range reporting. The total space complexity ends up being $O(s(n) \log^{d-2+\epsilon} n)$, and the total time complexity ends up being

$O(k + t(n)(\log n/\log\log n)^{d-2})$.

Chapter 3

Data Structures and Techniques

3.1 Augmented Cartesian Trees

The Cartesian Tree is a binary tree data structure. The nodes of this tree are ordered in a fashion that will be described shortly. This ordering, as we will see, allows for efficient range queries with minimal preprocessing. Multiple papers referred to in this thesis make use of the Cartesian Tree to supplement their range searching algorithms.

Consider a set of points $P = \{(x, y) \mid x, y \in \mathbb{R}\}$. For now, we can assume that no two points in P have the same x-coordinate or y-coordinate. This will allow us to capture the essence of the Cartesian tree without worrying about special cases. For this set of points P, the root of the corresponding Cartesian tree will be the point $(x_i, y_i) \in P$

with the minimum y-coordinate. The root of this point's right subtree will be the point (x_j, y_j) with minimum y-coordinate such that $x_i < x_j$. Inversely, the root of this point's left subtree will be the point (x_k, y_k) with minimum y-coordinate such that $x_i > x_k$. From here, a recursive definition follows. An arbitrary node N contains a point (x, y) as well as pointers to its left and right subtrees. N's left subtree is rooted by the point with minimum y-coordinate to the left of the N's point that hasn't already been added to the tree. Likewise, N's right subtree is rooted by the point with minimum y-coordinate to the right of N's point that hasn't already been added to the tree.

While the above recursive definition is fairly intuitive, it unfortunately doesn't provide an efficient method of construction. For each of the n nodes that get added, where $n = |P|$, we have to scan the original point set to find the ones with minimum y-coordinate to the left and right of the node. This yields an $O(n^2)$ construction time. Fortunately, an $O(n)$ construction time is provided by Gabow et al. [7] in Section 3 of their 1984 paper. This algorithm allows us to construct a Cartesian tree from P in a single pass of the list, and proceeds as follows. Assume that we already have a Cartesian tree T_{i-1} corresponding to the first $i-1$ points $\{p_1, ..., p_{i-1}\}$ in P as well as a pointer to the node containing p_{i-1}. To insert p_i into the tree, traverse from p_{i-1} up to the root of the tree until we reach a node containing p_k with y-coordinate less than that of p_i. From here, we insert a new node containing p_i in between p_k and its subtree. Gabow et al.'s algorithm takes advantage of the fact that Cartesian trees

also have the property that an inorder traversal yields the original list of points.

At first glance, it appears that the above construction algorithm is $O(n \log n)$. After all, we have n points being inserted into a binary tree. However, we can reason $O(n)$ worst-case running time by noticing the fact that we only ever traverse up the tree in constructing it, never downwards. To construct other types of trees, we have $O(\log n)$ traversals for each node. To insert a new node, we have to traverse potentially the entire tree to find the right spot to place the new node. But here, we have $O(\log n)$ traversals *in total*. In constructing the Cartesian tree, we only ever traverse in one direction. This yields a worst-case runtime of $O(n)$, since we have n nodes to insert in the tree.

To efficiently apply Cartesian trees to range searching, we will need to be able to quickly find the least common ancestor (LCA) of any two nodes. Without augmentation, we can find the least common ancestor of nodes A and B in $O(\log n)$ time by traversing up the tree from both nodes until we find the least common ancestor. However, to achieve the range searching time complexities touched on later in this paper, we will need a $O(1)$ time least common ancestor algorithm.

A naive approach to achieve this is via an $O(n^2)$ lookup table storing the pre-computed least common ancestors between every pair of points in the tree. However, Harel and Tarjan [6] were able to achieve a better result. In the end, using $O(n)$ preprocessing/space, they managed to achieve $O(1)$ time for finding the least common ancestor of two

nodes in an arbitrary tree. Their approach was as follows. They started by numbering the vertices in a clever way from left to right. Leaf nodes in the tree were numbered with odds $\{1, 3, 5, ...\}$ (starting with 1 and an increment of 2). The next layer was numbered starting with 2 and an increment of 4 $\{2, 6, 10, ...\}$. The layer after was numbered starting with 4 and an increment of 8 $\{4, 12, ...\}$. Thanks to this numbering, the height of the LCA above any two nodes could be found by a simple $O(1)$ bitwise XOR coupled with a log operation. For example, $depth\_LCA(1, 3) = \log(2) = 1$. After finding the depth of the LCA between two nodes, all that's needed is to report the nodes that's that height above the queried node.

As we can see, this alone provides an improvement; the lookup table size decreases to $O(n \log n)$. Instead of storing all pairs of points, we only need to store at most h entries for each node, where h is the height of the tree. After all, after finding the height of the LCA, we will only ever need to look up some height related to a node. For example, say we wanted to find the LCA between x and y. $depth\_LCA(x, y)$ will return a number h, signaling that the LCA is h nodes above x. We can index the $O(n \log n)$ lookup table with (x, h) to yield $LCA(x, y)$ in $O(1)$ time.

But with further preprocessing of time $O(n)$, Harel and Tarjan were able to construct a lookup table with $O(n)$ space. In the end, they came up with a way to preprocess the Cartesian tree (or any other tree) into an augmented Cartesian Tree. This is an $O(n)$ space data structure that allows the least common ancestor between any two

points to be queried in $O(1)$ time.

3.1.1 Three-Sided Range Reporting With Augmented Cartesian Trees

Sometimes, when we are querying a set of points, we might not want a lower bound for one of the features. Refer to the house example I gave above. I said "we want to find all houses 5-10 years old with a list price $500,000 and $600,000". But it might be more pertinent to query for houses 5-10 years old less than $600,000. The previous query could be filtering out a perfectly good house that's, say, $400,000.

In these cases, we'd want to use a three-sided range query. In a three-sided query, our query Q only has a tight bound on one of the features. The other feature only has another bound. In formal notation, $Q = [x_1, x_2] \times [-\infty, y_1]$.

A robust algorithm for 3-sided range reporting is alluded to in Section 2.1 of Alstrup et al [14]. It makes use of the augmented Cartesian tree elaborated on above. Given a set P of n points in $\mathbb{R}^2$, we could first process the set into a normal Cartesian tree. As mentioned above, this can be done in $O(n)$ time. Next, we augment this Cartesian tree in accordance with the common ancestor algorithm from Harel and Tarjan. This can also be done in $O(n)$ time.

However, we quickly run into a problem. Our end goal is to be able to find the point with minimum y-coordinate in the range $[x_1, x_2]$. This will provide us with not only a point to report as part of the query, but also a place to divide the query into two parts and recur. However, if we only include the n points $\in P$ in our Cartesian tree, we don't be able to do this in $O(1)$ time. Instead, we will have to traverse the tree until we reach the point with minimum y-coordinate $\in [x_1, x_2]$. This would take $O(\log n)$ time and render all the augmentation we did to find the LCA in $O(1)$ time pointless. To avoid this, Alstrup et al. considered three-sided queries only for points in the space $[N] \times \mathbb{R}$ rather than $\mathbb{R} \times \mathbb{R}$. Here, $[N]$ contains only non-negative integers $\{0, 1, ..., N-1\}$. N is the largest integer x-coordinate among all the points in $[N] \times \mathbb{R}$.

By doing this reduction to rank space, the above problem can be resolved. Instead of only adding the n points $\in P$ to our augmented Cartesian tree, we add another N points, one for each of the discrete ticks of the x-axis. Each of these "dummy" points has a y-coordinate of ∞, that way, no dummy point could be returned in a three-sided query as defined above. By including these dummy points, we have to incur extra space and preprocessing time: $O(N + n)$ instead of just $O(n)$. But in return, we have vertices directly corresponding to any $[x_1, x_2]$. By finding the least common ancestor of these vertices, which is guaranteed to be a non-dummy point by the above construction, we can report the point with minimum y-coordinate $\in [x_1, x_2]$ in $O(1)$. Let's say this point has x-coordinate x_a. To report the rest of the points in the original three-sided query, we can recur on $[x_1, x_{a-1}]$ and $[x_{a+1}, x_2]$, using the same

y_1 from our original query as the third side bound for both sub-queries. Using the same $O(1)$ LCA algorithm from above, we can report the next point $\in P \cap Q$ in $O(1)$. The recursion stops when the minimum point in $[x_i, x_j]$ has a y-coordinate greater than y_1. When this happens, there's no way that further subdivision of $[x_i, x_j]$ will yield any more points in the original query Q. In the end, the algorithm can perform any 3-sided query on a set of points $\in [N] \times \mathbb{R}$ in $O(k)$ time, where k is the number of points that lie in the query. This arises from the fact that there are k points to report and each point takes $O(1)$ time to report thanks to the extensive $O(N + n)$ space and time preprocessing of the points into an augmented Cartesian tree ahead of time.

Given points in $\mathbb{R} \times \mathbb{R}$, we can use the reduction to rank space technique on the x-coordinates to translate the set into points in $[n] \times \mathbb{R}$. In this new rank space, N from above is the same as n. Therefore, querying $[x_1, x_2] \times [-\infty, y_1]$ using the augmented Cartesian tree takes just $O(n)$ space and $O(k)$ time. Alstrup et al. [14] didn't explicitly mention this in their paper, but it follows directly from the concepts they laid out in their sections 2.1 and 2.2.

3.2 Range Trees

Like the Cartesian Tree, the Range Tree is also a binary tree data structure. It also allows for efficient range queries with minimal preprocessing and is mentioned in many of the papers referenced in this thesis. However, unlike the Cartesian Tree, the Range Tree generalizes to higher dimensions more easily. The augmented Cartesian Tree described above is tailored to handle 3-sided queries in a 2-dimensional space, but the Range Tree can be extended to d dimensions with relative ease. To see how this works, let's start by describing a range tree over points in one dimension.

Consider a set of points $P \in \mathbb{R}$. A range tree constructed over P will have all the points as leaf nodes. In the non-leaf nodes, we will store the largest x-coordinate contained in that node's left subtree. To construct such a tree, we can proceed in a recursive fashion. Let P_m be the median x-coordinate in P. Recursively construct two range trees: one on the points in P less than the median, another on the points in P greater than the median. These recursive calls, once implemented, will return the largest value in each of those subtrees. Connect these two subtrees to a parent node, and in this node, store the largest value in the left subtree. The recurrence relation for this construction is $T(n) = 2 * T(n/2) + O(n)$, which comes out to $O(n \log n)$. In the end, we use $O(n)$ space. The bottom layer has n nodes, the layer above that has $n/2$, and so on. $n + n/2 + n/4 + ... = 2n = O(n)$.

If P were stored as a simple list, then it would take $O(n)$ time to find all points $\in [a, b]$, where $[a, b]$ is a query range. This one dimensional range tree over P will allow us to perform this one-dimensional range query in $O(k + \log n)$, where k is the number of points that lie within the query range. Given $[a, b]$, we start at the root of the range tree, traversing down until we reach a node such that $a < node.value < b$. Call this node u. From u, we split, going right on one traversal path and left on the other. On the left path, once we hit a node, we go left if the node's value is $> a$, right otherwise, reporting all leaves in the right subtrees of visited nodes. Once we hit a leaf node, report it if it lies in the query range. Similarly, on the right path, once we hit a node, we go right if the node's value is $< a$, left otherwise, reporting all leaves in the left subtrees of visited nodes. Once we hit a leaf node, report it if it lies in the query range. Since the height of the range tree is $O(\log n)$, traversing the two paths takes $O(\log n)$ time. Reporting the k points as part of the query takes $O(k)$ time. Total time: $O(k + \log n)$.

The Cartesian Tree above contains one node per point. However, as described here, the Range Tree contains more nodes, storing metadata about each point. We can utilize this fact to extend the Range Tree to d dimensions. Consider a set of points $P \subset \mathbb{R}^d$. To construct a d dimensional range tree over P, first construct a 1-dimensional range tree over P_1, the first coordinate of each point in P. For each subtree in this 1-dimensional range tree, construct another range tree over P_2 for all points contained in that subtree. Set the corresponding node in the first tree to point to this next tree.

Continue in this fashion for all d dimensions. We can construct this d dimensional range tree in $O(n \log^{d-1} n)$ for $d \geq 2$. Each of the n points is contained in $O(\log^{d-1} n)$ subtrees, yielding a total space complexity of $O(n \log^{d-1} n)$. To query this tree for an orthogonal range $Q = [a_1, b_1] \times ... \times [a_d, b_d] \in \mathbb{R}^d$, we need $O(\log^d n + k)$ time. The general idea is centered around one key fact: a point P is only $\in Q$ if all d coordinates of P lie within the d ranges of Q. To query this d dimensional range-tree, we query the first dimension in $O(\log n)$ as described above. The subtrees that are a part of this query all point to subtrees in the next dimension. On each of the subtrees in the next dimension, we run the same two-path traversal to query them. From this, we get another set of subtrees, each pointing to subtrees in the next dimension. We continue this process for all d dimensions. It follows that for a two dimensional query, we need $O(\log^2 n + k)$, for a 3-dimensional query we need $O(\log^3 n + k)$, ..., and for a d-dimensional query we need $O(\log^d n + k)$ time.

3.3 Nested Interval Trees

In range searching algorithms, trees arise very frequently. They're memory-efficient, query-efficient, and intuitive. We've already seen two variants of trees so far, the Augmented Cartesian tree and the multi-dimensional Range tree. Ganguly et al. [2] makes use of another type of tree: the nested interval tree.

Given a set of intervals, how do we preprocess them so that we can quickly find the intervals that are stabbed by a certain point? A point x stabs an interval $[a, b]$ iff $a \leq x \leq b$. This is where the interval tree comes into play. The interval tree can be constructed in a recursive fashion. First, determine the center of the range covered by the union of all intervals. Store all intervals overlapping this center point in a node. This node points to all intervals completely to the left of the center point and also points to all intervals completely to the right of the center point. For both of these subsets of intervals, continue the same recursive algorithm.

By constructing the tree like this, we only need $O(n)$ space, where n is the number of intervals in the original set. Furthermore, we can get $O(\log n + k)$ query time, where k is the number of intervals that intersect a certain point. To query this tree, with a point p, first compare p with the "center" point mentioned above. If $p >$ "center", there's no way for any intervals in the left subtree to intersect p. Similarly, if $p <$ "center", there's no way for any intervals in the right subtree to intersect p. Essentially, at each level of the tree, we are eliminating half the intervals from consideration. We also have to report the k intervals that p does intersect with, yielding a $O(k + \log n)$ query time.

But what if intervals cannot be treated the same for the purposes of the problem at hand? It's absolutely possible that intersection with one interval means something different from intersection with another. For example, Ganguly et al. [2] consider two

different types of intervals: prev and next. They needed a way to distinguish between a point stabbing a "prev" interval and a point stabbing a "next" interval. This can be handled by extending the interval tree into a nested interval tree. Basically, the same principle that was applied to range trees is applied here. We create another "level" of trees, so that nodes on the first level not only point to their left and right children, but second-level subtrees. This allows us to differentiate between points intersecting "type 1" and "type 2" intervals. Similar to the multi-level range tree, this nested interval tree can be queried in $O(\log^2 n)$. As detailed later, the nested interval tree is an essential part of the 3-sided categorical range skyline query algorithm by Ganguly et al [2].

3.4 Reduction to Rank Space

In computational geometry, one often sees problems relating to points $\in \mathbb{R}^d$. Unfortunately for computer scientists, working in $\mathbb{R}^d$ often entails increased time complexity, space complexity, and edge cases.

Fortunately, we can often avoid these problems via a reduction to rank space. As Alstrup et al. [14] point out, we can reduce a general $\mathbb{R}^d$ range search problem to a simpler $[n]^d$ range search problem. Here, $[n] = \{0, 1, ..., n-1\}$. $[n]^d$ is referred to as "rank space". The algorithm to accomplish this is relatively straightforward. Consider a

point set $P \in \mathbb{R}^d$. For each of the d coordinates of points $p \in P$, there's some ordering of each of the coordinates. For example, imagine $P = \{(1,2), (\sqrt{2},4), (\pi,5)\} \in \mathbb{R}^2$. The ordering of the x-coordinates is as follows: $\{1, \sqrt{2}, \pi\}$. We can map these coordinates to $[n]^d$ as follows: $\{1, \sqrt{2}, \pi\} \to \{1,2,3\}$. In this fashion, the mapping $P \to \hat{P}$ is $\{(1,2), (\sqrt{2},4), (\pi,5)\} \to \{(1,1), (2,2), (3,3)\}$. Each coordinate of P is mapped to its rank among all coordinates to yield $\hat{P}$. By sorting P d times, one for each coordinate, we can perform this mapping in $O(dn \log n)$.

After performing a reduction to rank space, we avoid having to deal with $\mathbb{R}^d$. Instead, we can exploit the nice properties of the discrete point set $[n]^d$. This will allow for better space complexities and runtimes going forward.

3.5 The Grid Approach

As mentioned previously, we can perform orthogonal range queries in $d = 2$ dimensions in $O(\log^2 n + k)$ using a range tree taking $O(n \log n)$ space. Can we do better? As Alstrup et al. [14] show, we can. Making extensive use of the above reduction to rank space technique, Alstrup et al. managed to come up with a data structure that supports 2 dimensional range queries in $O(k + \log \log n)$. The general approach is as follows.

Consider M, a set of n points in $[n] \times [n]$, where $[n]$ is again a discrete axis marked by $\{0, 1, ..., n-1\}$. The goal is to process these n points into a recursive data structure. That way, given a query rectangle $[a, b] \times [c, d] \subseteq [u] \times [u]$, where $u \leq n$, we can answer this query efficiently by recursively discarding regions of $[n] \times [n]$ where there are no points in the query.

Here's a more formal description of the above. We start by dividing the grid with a set of row borders R and a set of column borders C. Between each pair of adjacent column borders, there are at most $\sqrt{n \log n}$ points by construction. Likewise, between adjacent row borders, there are also at most $\sqrt{n \log n}$ points by construction. There are at most $O(n/\sqrt{n \log n})$ columns, similarly, there at $O(n/\sqrt{n \log n})$ rows. Let $Q(i, j)$ denote the intersection of row i and column j, aka cell (i, j). We want to store each non-empty cell as a "point" in a top-level data structure. Furthermore, for each row and each column, we need to store a recursively defined data structure. For each level in the recursive structure, the points are translated to another rank space, only considering the points in that row/column. With this row/column grid and top-level structure established, there are now two possible cases for any query rectangle $[a, b] \times [c, d]$:

(a) The query spans multiple columns and multiple rows

(b) The query is completely inside only one column or only one row

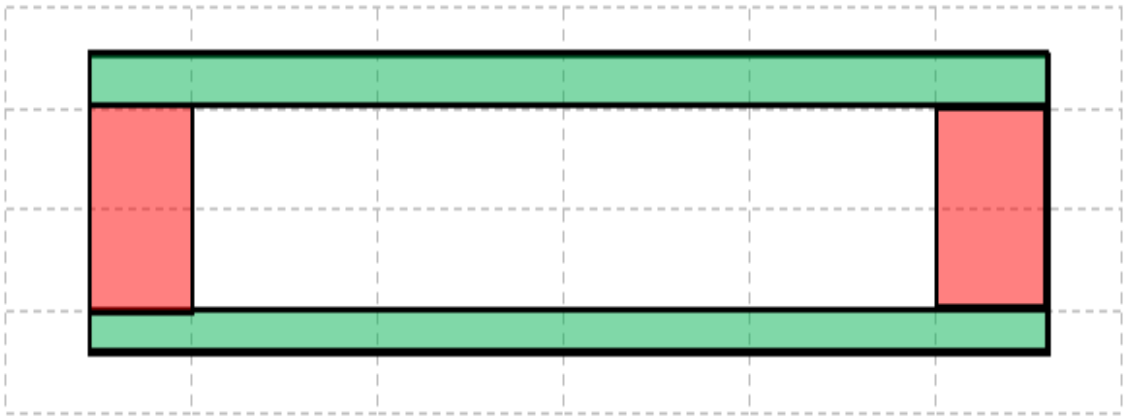


Figure 3.1: Case (a) of the grid approach. The green and red regions can be treated as 3-sided queries with respect to the middle region. To query the middle region, we can make use of our top-level data structure.

For case (a), we don't even need to perform any recursion. Instead, we can use row/column borders to divide the query into five regions. The four peripheral regions can be treated as 3-sided queries, an easier problem than 4-sided queries. 3-sided queries were defined and explored previously in this chapter. These 3-sided queries can be answered using a number of established algorithms in $O(\log \log n + k)$ time. After doing this, we are left with the one middle region. To answer this part of the query, we can make use of the top-level data structure described above. Each non-empty cell in the middle region is stored as a "point" in the top-level data structure. By querying the top-level structure, we can get all non-empty cells in the middle region. From there, we can just list off all the actual points in these non-empty cells. This is illustrated in figure 2.1 above.

For case (b), we can make use of the recursive data structure. If the query lies

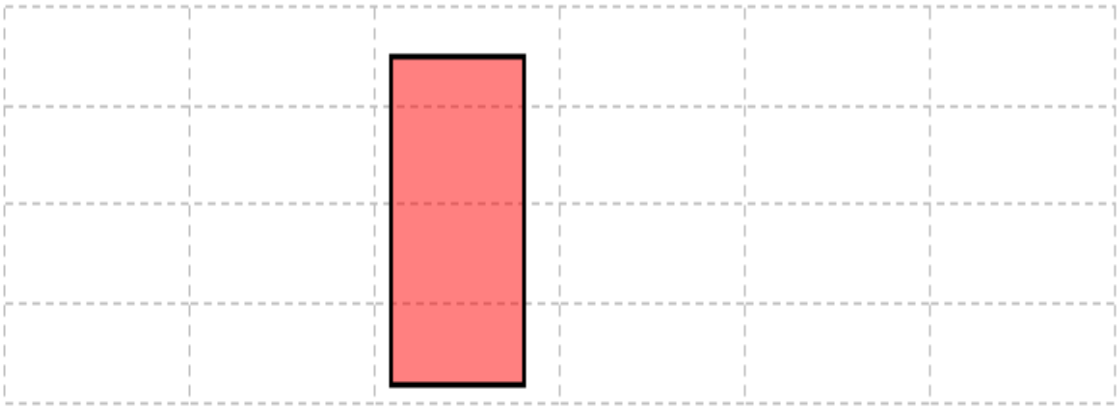


Figure 3.2: Case (b) of the grid approach. The query in red lies completely within a column. To query, we can recur on the data structure assigned to this column.

completely within a column, we can ignore all other columns in the original rank space; there's no way that any of those points are in the query. From here, we zoom in on this particular column, translating our query to the rank space of this column. Likewise, if the query lies completely within a row, we can ignore all other rows in the original rank space, then recur on that particular row. This is illustrated in figure 2.2 above.

On the surface, case (b) does not sound like good news for our space complexity. We start off with our original grid in $[n] \times [n]$ rank space. For each row and column in this grid that contains points, we have to create another grid in another rank space. For each row/column in *that* grid that contains points, we store yet *another* grid in another, more granular rank space. It sounds like we are dealing with exponential space. Fortunately, we are not. This follows from the above constraints. On the first

level, there are only n points total. By construction, there are at most $\sqrt{n \log n}$ points in a single row or column. As described above, we make a reduction to another rank space in that row/column, constructing more rows and columns for this new $[n_1] \times [n_1]$ rank space. $n_1 = \sqrt{n \log n}$ at most. By construction, there are at most $\sqrt{n_1 \log n_1} = \sqrt{\sqrt{n \log n} \log \sqrt{n \log n}}$ points in a single row/column on this next level. As you can see, the maximum number of points per row/column on successive recursion levels is decreasing rapidly in the form of an iterated square root. As Alstrup et al. note, on level $O(\log \log n)$, there's at most 3 points per cell. Therefore, we can stop the recursion and simply list the points in a row/column in constant time. The iterated square root essentially counteracts the "exponential" growth from earlier. All in all, the recursive structure can be stored in $O(n \log \log n)$.

However, there's still a question that needs to be answered. How do we determine whether $[a, b] \times [c, d]$ corresponds to case a or case b? In other words, we need some way to quickly find if $[a, b]$ straddles a column or $[c, d]$ straddles a row. This will take some time $q(n)$. As Alstrup et al. noted, $q(n) = O(\log \log u)$ is achievable by using the van Emde Boas Tree. This data structure is outlined in Peter van Emde Boas's [15] seminal 1975 paper. This ordered collection of (key, value) pairs supports methods findNext(k) and findPrevious(k). Given a value k, findNext(k) returns the smallest key $>$ k. Likewise, findPrevious(k) returns the largest key $<$ k. Both methods run in $O(\log \log u)$, where u is the size of the universe spanned by the set of points. We can make use of this data structure to quickly differentiate between case a and case

b. Given $[a, b]$, we can query a vEB tree containing all column borders with a then b to find out if $[a, b]$ straddles a single column in $O(\log \log n)$. Similarly, we can query a vEB tree containing all row borders with c then d to find out if $[c, d]$ straddles a single row in $O(\log \log u)$.

Later in their paper, Alstrup et al. detailed how they were able to do even better than $q(n) = O(\log \log n)$ for differentiating between case a and case b. Finding whether $[a, b]$ straddles a column border or $[c, d]$ straddles a row border is an instance of the *existential* 1-dimensional range query. This is because we can treat column borders as a set of points C along the x-axis. Likewise, we can treat row borders as a set of points R along the y-axis. Given range $[a, b]$ on a discrete axis, does it contain any points in C? Existential range queries like these were considered by Miltersen et al. [11]. Miltersen et al. study the existential range query problem in a different model of computation. Instead of viewing the problem as queries to a data structure, they model it as communication between two parties, denoted Alice and Bob. Alice has a query interval $[a, b]$, Bob has a set of points S, and they have to determine if $[a, b] \cap S = \emptyset$. To achieve an $O(1)$ query time, Miltersen et al. first notice that given two numbers a, and b, they have a common prefix in their bit representations. Call this prefix w. Alice sends w to Bob, which Bob hashes, returning the smallest element $\in S$ with bit prefix w. Bob sends this smallest element back to Alice. Alice knows that if $b <$ the smallest element in S with prefix w, then $[a, b] \cap S = \emptyset$. As detailed in the paper, this entire process takes place in $O(1)$ time.

With Miltersen et al.'s [11] data structure in play, Alstrup et al. were able to achieve $O(k + \log\log n)$ range queries for $[a, b] \times [c, d]$ queries on n points in $[n] \times [n]$. Despite the complexity of Alstrup et al.'s algorithm, not only is the query time very fast, but the space complexity is modest as well. As Alstrup et al. describe, their data structure uses $O(n \log^{\epsilon} n)$, where $\epsilon > 0$.

Chapter 4

The Range Skyline Problem

As mentioned previously in the section on Orthogonal Range queries, we often want to answer a query Q on a set of points $P \subset \mathbb{R}^d$. However, sometimes, we want to focus only on the points in Q that have certain characteristics. In the general Range Skyline problem, we are looking for the points $\in Q$ not dominated by any other point $\in Q$. In 2 dimensions, a point $p1 = (x1, y1)$ dominates a point $p2 = (x2, y2)$ if $x1 \geq x2$ and $y1 > y2$ or $x1 > x2$ and $y1 \geq y2$. For $p1$ to dominate $p2$, the value of $p1$ must be at least the value of $p2$ in one dimension and must be strictly larger in the other dimension. Some papers use the inverse definition of dominance, where $p1 = (x1, y1)$ dominates a point $p2 = (x2, y2)$ if $x1 \leq x2$ and $y1 < y2$ or $x1 < x2$ and $y1 \leq y2$. This concept of point dominance can be extended to d dimensions, as it is in some variants of the range skyline problem.

In this thesis, we will touch on several variants of the range skyline problem. Rahul et al. [12] consider cases where points in set P have each two different coordinates in two different spaces. Ganguly et al. [2] construct their range skyline in the I/O model of computation. In Ganguly et al.'s paper, each point in P is assigned a different color. A new variant of the problem that Dr. Nekrich and I have been working on makes use of the concepts outlined in these papers.

4.1 Orthogonal Range Skyline Queries

Given a set of points P, each point has coordinates in $\in \mathbb{R}^d$ (denoted "range space") as well as $\in \mathbb{R}^t$ (denoted "feature space"). What's the best way to preprocess these points in order to answer skyline queries in feature space? Rahul et al. [12] sought to answer this question.

We can extend the aforementioned home buying example to provide intuition for this problem. Instead of each home being described by just $(price, age)$, let's say that each home has two coordinate pairs. $(latitude, longitude)$ in range space, and $(price, age)$ in feature space. When speculating in the housing market, we might only want to consider only the skyline of the set of homes. There's no reason to consider a home that costs more and is older than another; only the dominant homes matter in our home buying search. However, location matters as well. If we were to look

at the skyline among all homes, we could have places with $(lat, long)$ coordinates thousands of miles away from where we want to live. To account for this, we want to consider the skyline of homes that lie within a certain orthogonal box in range space: $[lat_1, lat_2] \times [long_1, long_2]$. A naive approach would be to sort the entire list of homes and iterate through, methodically adding ones that lie in $[lat_1, lat_2] \times [long_1, long_2]$ and the feature space skyline. However, this would require $O(n(d+t))$ query time. Recall that d is the number of dimensions in range space while t is the number of dimensions in feature space. Rahul et al. sought to do better.

Rahul et al. [12] constructed a range skyline query structure based on the range tree. This time, instead of being just d-dimensional, this range tree is $\delta = d+t$-dimensional. Furthermore, this range tree is augmented with extra information on its last level. Given feature space coordinates $(a_1, ..., a_t)$, the last level of the range tree stores the point with minimum a_t coordinate. By Rahul et al.'s definition of point dominance, this point cannot be dominated by any other, and therefore must be in the skyline.

Rahul et al.'s augmented range tree allows for $O((k+1)\log^{\delta} n)$ queries. Given query Q, an axes-parallel box in range space, we can apply the range tree query algorithm from above. After traversing through the first d levels of the tree, we end up at the feature space part of the tree. At first, we can't rule any part of feature space out. However, the extra information we stored allows us to start refining our search. We know the point with minimum a_t coordinate is in the skyline. This not only allows

us to add this point to the skyline, but also rule out parts of feature space that can't possibly contain the skyline. For example, we can rule out the box with feature space coordinates less than a_t. As Rahul et al. show in their paper, they managed to partition the remaining space into disjoint boxes. Following this partition, we can recur on each of the feature subspaces. For each space, we again traverse through the range tree, partitioning this feature subspace around the point with minimum a_t coordinate within. For each of the k points to be reported, we traverse through the full δ-dimensional range tree. We also have some partial traversals corresponding to searching feature subspaces that contain no points. As Rahul et al. note, the total query time becomes $O((k+1)\log^{\delta} n)$. As in a standard range tree, space and construction time are both $O(n \log^{\delta-1} n)$.

4.2 I/O Optimal Categorical 3-sided Skyline Queries

Here, we consider categorical range skyline queries. In categorical range skyline queries, each point in P is assigned a category. Ganguly et al. [2] refers to categories as "colors". The goal of a categorical range skyline query, rather than returning the entire skyline within a query, is to report the different colors of points on the skyline. For example, say we had a skyline consisting of two blue points, two green points,

and two red points. To answer a categorical range skyline query, we'd only report "blue, green, red". An obvious solution is to use a regular skyline query algorithm, similar to the one Rahul et al. [12] detailed, then just iterate through the skyline, reporting the different colors. Multiple points in the skyline can have the same color. In the above example, to report only three colors, we'd have to look at six points. Ganguly et al. aimed to tackle this inefficiency.

Ganguly et al. [2] also designed their algorithms to work in the I/O model of computation. Recall that the RAM model assumes $O(1)$ time for load/store, while the I/O model doesn't. In the I/O model, queries to the external memory take $O(B)$ time, where B is the word size. In the I/O model, there's even more incentive not to use regular range skyline algorithms to answer categorical range skyline queries. To 'offset' the added complexity induced by the I/O model, Ganguly et al. limit themselves to 3-sided queries. In many cases, 3-sided queries tend to be easier to efficienty handle then 4-sided ones. While not explicitly stated above, this was alluded to in the description of Alstrup et al.'s 3-sided and 4-sided range query results [14]. The 4-sided query algorithm was significantly more complicated than the 3-sided query algorithm.

With preliminaries out of the way, we can begin describing how Ganguly et al. managed to create an $O(n \log^* n)$ space data structure to answer 3-sided categorical range skyline queries in $O(1 + k/B)$ I/Os. First, to simplify construction of their data

structure, Ganguly et al. assumed that all points can be reduced to $n \times n$ rank space. Then, they focused on 3-sided queries of the form $[a, b] \times [\tau, \infty]$. Note that this is slightly different from the "upside down" queries referred to in the previous chapter. Alstrup et al. [14] used an Augmented Cartesian Tree for queries of the form $[x_1, x_2] \times [-\infty, y_1]$. But Ganguly et al.'s approach fits better with $[a, b] \times [\tau, \infty]$ queries. Next, for each point $(i, A[i], C[i])$, where i is the x-coordinate of the point in rank space, $A[i]$ is the y-coordinate, and $C[i]$ is the point's color, two parameters were defined as follows:

1. $prev(i) = max(j) \in [1, i)$ such that

   (a) $C[i] = C[j]$ (same color)

   (b) $A[j] > A[i]$ (greater y-coordinate)

   (c) $(j, A[j])$ isn't dominated by any $(k, A[k])$, where $j < k \leq i$

   If no such j exists, set $j = -\infty$

2. $next(i) = min(j) \in (i, n]$ such that

   (a) $A[i] < A[j]$

   If no such j exists, set $j = \infty$

Assigning two numbers like these to each point is a technique known as "double chaining". Each point in the set is linked to two others. This technique allows us to

establish multiple lemmas that will be useful later on:

1. Given $[a, b] \times [-\infty, \infty]$, we can immediately say $(i, A[i], C[i])$ is on the skyline of this region if $next(i) > b$, where $next(i)$ is the next point that dominates i. If it is outside the query region and has the same color, then i must be on the skyline.

2. Given $[a, b] \times [-\infty, \infty]$, i is the highest point with color $C[i]$ on the skyline if $prev(i) < a$, where $prev(i)$ is the closest point to the left that is higher than i and has the same color.

3. Given $[a, b] \times [\tau, \infty]$, point i is on the skyline if $(prev(i), i]$ is stabbed by a and $[i, next(i))$ is stabbed by b.

The third lemma in particular is quite useful for the algorithm of Ganguly et al. [2]. By rephrasing the problem in terms of interval stabbing, they were able to make use of the nested interval tree, described previously. All backward intervals of the form $(prev(i), i]$ can be processed into a first-level interval tree. Then, another level containing intervals of the form $[i, next(i))$ can be layered on top of that first level. By doing so, we can find the intervals stabbed by a and b in $O(\log^2 n)$ time. To obtain I/O optimality, we can make a further optimization by condensing the bottom $\log B$ levels of the trees, taking advantage of locality to yield $O(\log^2(n/B))$ I/Os.

However, this is only a part of the picture. We've found all intervals stabbed by a

and b, but there's a third-element to three sided queries: τ. To proceed, Ganguly et al. noticed that this can be reduced to a problem of reporting point dominance in 3-dimensions. They then made use of Afshani's I/O efficient data structure to report point dominance in 3D [1]. Afshani's data structure uses linear space and can be queried in $O(\log(n/B)+k/B)$. In conjunction with the nested interval tree here, that brings the total 3-sided categorical range skyline query I/Os to $O(\log^3(n/B)+k/B)$.

Noticing that $\log^3(n/B)$ is a low number even for large values of n, Ganguly et al. [2] sought to achieve $O(1+k/B)$ query I/Os. To this end, they crafted a method of efficiently storing precomputed solutions. Ganguly et al. refer to their method as "bootstrapping".

Ganguly et al. [2] took a creative approach, storing two different types of precomputed solutions. At a high level, type 1 precomputed solutions are stored from left to right, while type 2 precomputed solutions are stored from right to left. Now, consider $[a,b]$ in query range $[a,b]\times[\tau,\infty]$. Assume for now that we know the size of the solution, k, ahead of time. We can then partition $[a,b]$ into several different subintervals. Then, we can assemble the above precomputed solutions in $O(1+k/B)$ I/Os.

The key here is to define the intervals such that the previous $O(\log^3(n/B)+k/B)$ nested interval tree algorithm can be applied here as well. We want n to be small enough such that $O(\log^3(n/B)+k/B) \rightarrow O(1+k/B)$. Ganguly et al. managed to accomplish this. However, a balance must be found; make n too small and the

number of precomputed solutions that must be stored increases, increasing space complexity as well. Ganguly et al. explored this space/time tradeoff, constructing the $O(n \log^* n)$ space, $O(1 + k/B)$ query I/Os algorithm above, but also an $O(n)$ space, $O(\log^* n + k/B)$ query I/Os one.

Above, we noted how Ganguly et al. assumed the value of k ahead of time in constructing their bootstrapping-based solutions. At first glance, this seems like an assumption that significantly reduces the problems complexity. But it turns out that, asymptotically, converging on the correct value of k adds nothing to the above query I/Os. As detailed in the paper, one can make use of a for loop, essentially looping through powers of 2 until the correct value of k is found. This only adds $O(1 + k/B)$ I/Os, adding nothing to the above asymptotic I/O complexity.

Chapter 5

New Work on Skyline Queries

As you can see, the orthogonal range skyline problem has been studied in a number of previous papers. However, the problem could not at all be considered completely solved. With all the preliminaries in place, we're now in a position to describe the contributions of this thesis, then explore potential future research directions.

5.1 Novel Reduction from 3-sided Skyline Point Queries to 3-sided Orthogonal Point Reporting

Given a point set P in $n \times n$ rank space, how can we efficiently answer 3-sided skyline queries of the form $[a, b] \times [c, \infty]$? Well, in the general case, a point $p \in P$ belongs to the skyline of this query if

(a) $p.x \geq a$

(b) $p.x \leq b$

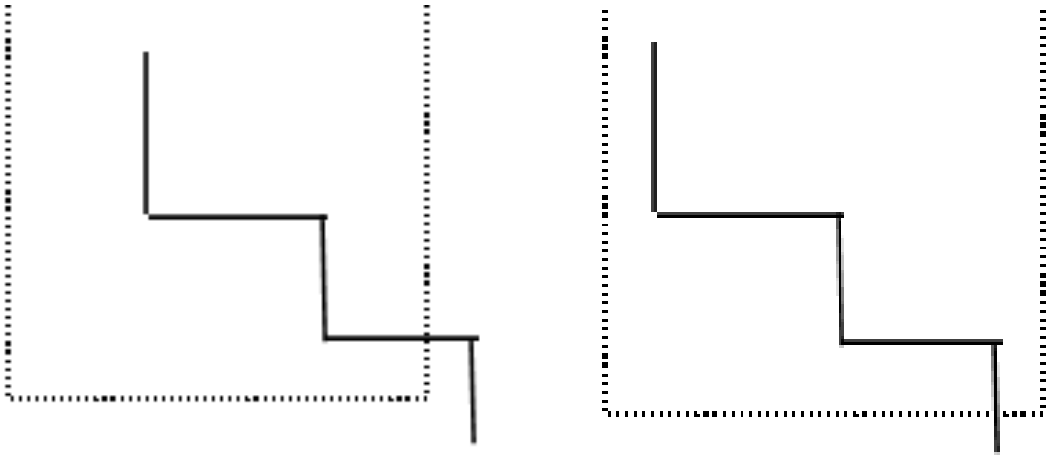
(c) $p.next > b$

(d) $p.y \geq c$

The above $p.next$ was defined in Section 4 in reference to Ganguly et al. [2]. For reference, $next(i) = min(j) \in (i, n]$ such that $A[i] < A[j]$ for a point $(i, A[i])$. Therefore, we can reduce the 3-sided skyline query problem to an orthogonal range query in 3d space. The new 3d space here is $p.x \times p.y \times p.next$. In this space, we're querying the half open 3d box $[a, b] \times [c, \infty] \times [b, \infty]$. Data structures do exist to answer half open 3d box queries like this, but the algorithms for querying half open 2d boxes are more

efficient. For example, Alstrup et al. [14] managed to achieve $O(k)$ query time for 3-sided point reporting by preprocessing their points into the Augmented Cartesian Tree described previously.

With this in mind, we can eliminate one of the above conditions. Notice that there are two possible cases for skyline queries, illustrated in the figures below.



(a) Case 1: The skyline is entirely above $y = c$

(b) Case 2: The skyline dips below $y = c$

Figure 5.1: Two cases of 3-sided skyline queries

We don't need all of the 4 conditions above when dealing with Case 1 queries. Since the entire skyline lies above $y = c$, we don't need to consider the $p.y \geq c$ condition. Since only 3 conditions remain, we can reduce the skyline query to an orthogonal range query in the space $p.x \times p.next$. In this new space, we're querying the half open 2d box $[a, b] \times [b, \infty]$. This can be done using an efficient data structure for 3-sided orthogonal range queries such as the one described by Alstrup et al. [14]. However,

Case 1 queries aren't the only possible type; Case 2 queries exist as well. But if we can find some $b' < b$ such that $[a, b'] \times [c, \infty]$ is a Case 1 query, we can apply the reduction to the 3-sided orthogonal range query as described above.

Theorem 1 *Let* $b' = max\{p.x | p.y > c$ *and* $p.next > b\}$. *For any point* p, p *is on the skyline of the query range* $[a, b] \times [c, \infty]$ *iff* $a \leq p.x \leq b_1$ *and* $p.next > b$, *where* $b_1 = min(b, b')$.

Proof: The above theorem is in biconditional form, so we must prove both implications. Let's first assume that $a \leq p.x \leq b_1$ and $p.next > b$; must p be on the skyline? Well since $p.next > b$, the next point that could dominate p is outside of the query range by definition of the *.next* pointer. And $a \leq p.x \leq b_1$, so p is within the x-coordinate bound of the query range. Furthermore $p.y > c$, since that's part of the definition of b'. Point p meets the original 4 conditions above. So if $a \leq p.x \leq b_1$ and $p.next > b$, then p must be on the skyline.

Next, we assume that p is on the skyline of the query range. Does $a \leq p.x \leq b_1$ and $p.next > b$ hold? Well again, by definition of the *.next* pointer, p being on the skyline immediately implies that the next point dominating p must be outside of the query range. Since p is in the query range, it immediately follows that $a \leq p.x$. But what if $b_1 \leq p.x \leq b$; is this possible? Not according to the definition of b_1. Recall that $b_1 = min(b, b')$, where $b' = max\{p.x | p.y > c$ and $p.next > b\}$. If $b_1 \leq p.x \leq b$

held for a point p on the skyline, that would contradict the definition of b'. This point p, since it's on the skyline, would have $p.y > c$ and $p.next > b$. But somehow, this $p.x > MAX\{p.x | p.y > c \text{ and } p.next > b\}$? That's a contradiction, meaning that $b_1 \leq p.x \leq b$ cannot hold for any point p on the skyline.

□

With b' formally defined, we note that b' can be found efficiently in $O(\log \log u)$ using a modified form of Chan's [3] data structure, based on the vEB tree mentioned above. As was mentioned above, u is the size of the universe that the points lie in. Here in rank space, that universe is of size n.

5.2 Counting Colors in a Skyline

But what if points in P have different colors? Is there a way for us to efficiently report the different colors of the skyline of query $Q = [a, b] \times [c, \infty]$ in $n \times n$ rank space? What about the number of points of each color? We could just use the above algorithm to get the skyline of Q, then count the number of different colors. However, is there a better way?

Much work has been already done for counting in orthogonal range queries , such as Chan et al. [4]. Chan et al. defined and explored two different types of orthogonal

range counting: type 1 and type 2. The type 1 counting problem is to find the number of distinct colors in the query range. The type 2 counting problem expands on this, asking for the number of distinct colors AND the number of points of each color. Our skyline problem is the more difficult type 2 counting problem. Fortunalely, Chan et al. [4] described how to answer a type 2 orthogonal range counting query in $O(\frac{\log n}{\log \log n} + k \log \log n)$, where k is the number of colors in the range.

Making use of Chan et al.'s [4] $O(\frac{\log n}{\log \log n} + k \log \log n)$ orthogonal range colored point counting algorithm, we can finally outline the specifics of our new $O(\frac{\log n}{\log \log n} + k \log \log n)$ skyline colored point counting algorithm. Given a set of points P and 3-sided skyline query range Q, we can start by performing the reduction mentioned in section 5.1. This transforms the original skyline colored point counting problem into an orthogonal range colored point counting algorithm. Again, the orthogonal range is in a different space than the original coordinates; the original coordinates are in $p.x \times p.y$ space while this new orthogonal range is in $p.x \times p.next$ space. By using Chan et al.'s orthgonal range colored point counting algorithm on the transformed query in $p.x \times p.next$ space, we can answer the original skyline colored point counting query in $O(\frac{\log n}{\log \log n} + k \log \log n)$.

Theorem 2 *For query $Q = [a, b] \times [c, \infty]$ on a set of points P, there's a data structure that can report the number of points of each color on the skyline of Q, taking $O(\frac{\log n}{\log \log n} + k \log \log n)$ to do so. k is the number of colors in the range. This data*

structure can store the set of colored points in $O(n \log \log n)$ words of space.

5.3 Potential Research Directions

Ganguly et al.'s [2] algorithm, while robust, is quite complicated. We could simplify it by applying the above b' reduction, reducing the 3-sided categorical skyline query to a 3-sided orthogonal range query. By reducing the categorical skyline query to a categorical 3-sided orthogonal range query, we could make use of the simpler algorithms that already exist for 3-sided categorical orthogonal range queries. For example, Patil et al. [9] were able to answer categorical 3-sided queries in $O(1 + k/B)$ I/Os. This is the same time complexity that Ganguly achieved. But since Patil et al. designed their algorithm for 3-sided categorical *orthogonal* range queries, their algorithm is noticeably simpler.

We also might be able to use something similar to Alstrup et al.'s [14] grid approach for answering 3-sided skyline queries. Recall from above, our set of points P are in $n \times n$ rank space, not $\mathbb{R}^2$. Therefore, our 3-sided $[a, b] \times [c, \infty]$ could be seen as a 4-sided query, $[a, b] \times [c, n]$. Since Alstrup et al. made use of the grid approach to answer 4-sided 2-dimensional orthogonal range queries, we might be able to make use of a similar grid approach to answer 4-sided 2-dimensional skyline queries.

As Alstrup et al. [14] did, let's divide our set of points into rows and columns, each containing at most $\sqrt{n \log n}$ points. Just like Alstrup et al. described, each row and column contains a recursively defined data structure with a new rank space. The difference between our grid approach and Alstrup et al.'s lies in the top-level data structure. To handle case (a) queries (not completely within a row or column), Alstrup et al. reported all "points" in their top level structure, where each "point" corresponds to a non-empty cell (intersection of a row and column). From there, they reported each actual point corresponding to the top-level "point", thus answering the original query. Creating the mapping between top-level "points" and actual points was a one-time cost in preprocessing.

In our top-level structure, each "point" again corresponds to a non-empty cell. But this time, we don't want to store *all* actual points corresponding to a top-level "point". Rather, we just want the skyline of that cell. Fortunately, this is just a one-time cost paid in preprocessing. To answer a case (a) query, we find the skyline of the top-level "points", then, for each "point" in the top-level skyline, we can get the actual points in the skyline of that cell. Answering the query is then a matter of merging the skylines of each cell.

Originally, we have n points. By construction, we have $O(n/\sqrt{n \log n})$ columns and $O(n/\sqrt{n \log n})$ rows. Therefore, we have $O(n/\sqrt{n \log n}) * O(n/\sqrt{n \log n}) = O(n/\log n)$ "points" in the top-level structure. By applying this modified version

of Alstrup et al.'s [14] grid approach, we've reduced the problem from finding a skyline among n points to finding a skyline among $O(n/\log n)$ "points" in the top-level structure. That's a significantly smaller set of points. All of the above suggests we could apply something similar to Alstrup et al.'s [14] grid approach to efficiently answer skyline queries.

5.4 Open Questions

Despite all the work that's already been done on skyline queries and the work that's been contributed by this thesis, there are still several open questions in the field. For example, we know that we could, in theory, perform the section 5.1 reduction in constant time. Since b' is a function of only c and b as described above, we could precompute a lookup table that takes a (c, b) pair as input and outputs the correct b'. However, a naive form of this lookup table would take $O(n^2)$ space; untenable for even moderately large n. Could we reduce the space usage while still keeping the time complexity constant? Perhaps we could make use of a space-saving technique like Harel and Tarjan [6] did in their $O(1)$ LCA tree augmentation technique, but this remains an open question for now. If we could perform the section 5.1 reduction in constant time without using too much space, then 3-sided skyline queries would be reducible to 3-sided orthogonal range queries in $O(1)$. Since 3-sided orthogonal range queries can be answered in optimal $O(k)$ time using Alstrup et al.'s [14] Cartesian

Tree, this would mean an optimal solution for the 3-sided skyline query.

Other open questions pertain to that of colored skyline reporting and counting, described above. They also hinge on the efficiency of the section 5.1 reduction. Since we already have efficient algorithms for colored orthogonal range reporting and counting, if we could efficiently reduce skyline queries to orthogonal range queries, we could make use of these orthogonal range reporting query algorithms.

It might also be possible to make use of the modified grid approach I described above. Alstrup et al.'s [14] original grid approach offered $O(k + \log\log n)$ query time; it follows that a skyline algorithm based on this grid approach would be of a similar order of efficiency. However, I haven't yet formalized this approach, proving a complexity bound. Furthermore, I still have to work out how to efficiently merge the skylines contained within top-level "points". If these questions are answered, this grid approach could provide us with near-optimal time complexity for 3-sided skyline queries.

References

[1] Peyman Afshani. On dominance reporting in 3d. *16th Annual European Symposium*, pages 41–51, 2008.

[2] Sharma V. Thankachan Arnab Ganguly, Daniel Gibney and Rahul Shah. I/o-optimal categorical 3-sided skyline queries. *Theoretical Computer Science*, 896:132–144, 2021.

[3] Timothy M. Chan. Persistent predecessor search and orthogonal point location on the word RAM. *ACM Trans. Algorithms*, 9(3):22:1–22:22, 2013.

[4] Timothy M. Chan, Qizheng He, and Yakov Nekrich. Further results on colored range searching. In Sergio Cabello and Danny Z. Chen, editors, *36th International Symposium on Computational Geometry, SoCG 2020, June 23-26, 2020, Zürich, Switzerland*, volume 164 of *LIPIcs*, pages 28:1–28:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[5] Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal on Computing*, 15(3):703–724, 1986.

[6] Dov Harel and Robert Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, 1984.

[7] Robert Tarjan Harold Gabow, Jon Louis Bentley. Scaling and Related Techniques for Geometry Problems. *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 135–143, 1984.

[8] Joseph F. JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In Rudolf Fleischer and Gerhard Trippen, editors, *Algorithms and Computation, 15th International Symposium, ISAAC*, volume 3341 of *Lecture Notes in Computer Science*, pages 558–568. Springer, 2004.

[9] Manish Patil, Sharma V. Thankachan, Rahul Shah, Yakov Nekrich, and Jeffrey Scott Vitter. Categorical range maxima queries. In Richard Hull and Martin Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 266–277. ACM, 2014.

[10] Mihai Patrascu. Lower bounds for 2-dimensional range counting. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC

'07, page 40–46, New York, NY, USA, 2007. Association for Computing Machinery.

[11] Shmuel Safra Peter Miltersen, Noam Nisan and Avi Wigderson. On data structures and asymmetric communication complexity. *Journal of Computer and System Sciences*, 57(1):37–49, 1998.

[12] Saladi Rahul and Ravi Janardan. Algorithms for range-skyline queries. *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, page 526–529, 2012.

[13] Donald Kossmann Stephan Börzsönyi and Konrad Stocker. The skyline operator. *IN ICDE*, pages 421–430, 2001.

[14] Theis Rauhe Stephen Alstrup, Gerth Stølting Brodal. New Data Structures for Orthogonal Range Searching. *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 198–207, 2000.

[15] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. *16th Annual Symposium on Foundations of Computer Science*, pages 75–84, 1975.